ANÁLISIS DEL RETO

Integrantes individuales:

Adriana Sofia Rozo Cepeda Cod. 202211498 (as.rozo@uniandes.edu.co)
Diego Fernando Galván Cruz Cod. 202213709 (d.galvanc@uniandes.edu.co)
Juan David Castillo Quiroga Cod. 202210669. (j.castilloq@uniandes.edu.co)

Requerimientos grupales básico:

1. Requerimiento 1: Encontrar los videojuegos
   Publicados en un rango de tiempo para una plataforma.
2. Encontrar los 5 registros con menor tiempo
   Para un jugador en específico.

Requerimientos individuales:

3. Diego Fernando Galván Cruz
4. Juan David Castillo Quiroga
5. Adriana Sofia Rozo Cepeda

Requerimientos avanzados:

6. Requerimiento 6: Diagramar un histograma de
   Propiedades para los registros de un rango de años.
7. Requerimiento 7: Encontrar el TOP N de los videojuegos más rentables para retransmitir.

análisis de complejidad → O() + Tiempo de ejecución: Pruebas realizadas en máquina 1.

Espacio de la carga [KB]

Porcentaje de la muestra [pct]	Carga
5.00%	5722.34
10.00%	10488.33
50.00%	43035.87
80.00%	65290.59
100.00%	79701.58

Ambientes de prueba:

	Maquina 1	Maquina 2	Máquina 3
Procesadores	11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz	**11th Gen Intel(R) Core (TM) i7-1165G7 @ 2.80GHz 2.80 GHz**	Apple M1 Chip
Memoria RAM (GB)	16,0 GB (15,7 GB usable)	16,0 GB (15,7 GB usable)	8 GB Memoria unificada
Sistema Operativo	Sistema operativo de 64 bits, procesador basado en x64	**Windows 11 Home Single Language. 64-bit operating system, x64-based processo**	**macOs**

Máquina 1:

Porcentaje vs Tiempo

Porcentaje de la muestra	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
5.00%	**5.32**	**0.58**	**1.18**	**11.49**	**1.67**		
10.00	**100.62**	**1.36**	**55.97**	**3813.55**	**366.78**		
50.00	**143.39**	**2.01**	**2583.01**	**117665.74**	**13307.44**		
80.00	**146.13**	**3.88**	**7109.21**	**604498.1**	**44916.36**		
100.00	**141.19**	**6.33**	**11758.98**	**802992.11**	**90110.49**		

Porcentaje vs memoria

Porcentaje de la muestra	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
5.00%	**45.02**	**4.53**	**39.19**	**23.29**	**2.05**		
10.00	**65.45**	**4.07**	**5.94**	**10.57**	**6.14**		
50.00	**65.89**	**6.88**	**6.53**	**12.15**	**5.13**		
80.00	**65.57**	**5.9**	**6.02**	**7.67**	**11.97**		
100.00	**15.35**	**0.52**	**4.16**	**-**	**64.04**		

Máquina 2:

Porcentaje de la muestra	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
5.00%							
10.00							
50.00							
80.00							
100.00							

Porcentaje de la muestra	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
5.00%							
10.00							
50.00							
80.00							
100.00							

Máquina 3:

Tiempo

Porcentaje de la muestra	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
5.00%	**118.2**	**3.59**	**65.37**	**2.52**	**105.51**	**462.42**	**582.24**
10.00	**151.03**	**3.34**	**128.41**	**3.56**	**620.53**	**1642.48**	**1744.92**
50.00	142.12	2.30	2353.62	117842.29	12483.39	**13525.28**	**14274.31**
80.00	147.20	3.1	7251.47	605724.12	500452.49	**793232.42**	**801314.57**
100.00	141.52	5.97	11215.27	800451.48	901462.49	**1367428.42**	**14658243.27**

Memoria

Porcentaje de la muestra	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
5.00%	**24.28**	**4.99**	**51.21**	**6.12**	**10.43**	**62.41**	**60.23**
10.00	**64.34**	**4.14**	**6.01**	**4.59**	**7.44**	**9.42**	**12.85**
50.00	68.22	5.47	6.23	12.15	5.32	**9.92**	**13.26**
80.00	68.85	6.20	5.97	7.67	12.02	**15.64**	**16.96**
100.00	14.87	1.02	4.82	8.46	64.51	**68.42**	**70.61**

Requerimiento 1:

O(n^2)

En el requerimiento 1 en la variable de map1 hacemos alusión a el analyzer donde está el índice con la carga de datos para nuestro requerimiento 1. La complejidad de llamar esta carga podría resumirse en ella misma. La complejidad para la carga de datos es de 0(n) ya que la última instrucción es que, si la llave existe, obtenga el valor y lo añada a la lista.
En nuestro requerimiento extraemos los valores del árbol, seguido a esto creamos una lista tipo ARRAY_LIST. Iteramos los valores del árbol para extraer los juegos y seguido a esto iteramos cada juego para obtener los datos de cada columna.
Si el juego cumple con la condición dada por el if después del for se añade el verbo a la lista.
Lo último que realizamos es organizar dicha lista con insertion sort y extraemos los primeros y últimos.

```
#=^..^=   [Requerimiento 1]  =^..^=    =^..^=    =^..^=    =^..^=

def Req1_VideogamesByRangeDate(analyzer, platform, min_date, max_date):
    map1 = analyzer["Req1_VideogamesByRangeDate"]
    lst = om.values(map1, min_date, max_date)                    #O(logn)
    games_by_platform = lt.newList(datastructure = "ARRAY_LIST")  #O(1)

    for lst_games in lt.iterator(lst):                           #O(n^2)
        for game in lt.iterator(lst_games):
            if platform in game["Platforms"]:
                lt.addLast(games_by_platform, game)              #O(n^2)

    sorted_list = sortList(games_by_platform, cmp_Req1)          #O(n^2)
    final_list = FirstandLast(sorted_list)                       #O(n^2)
    return final_list, listSize(sorted_list)                     #La complejidad se resume en el ordenamiento con insertion O(n^2)
```

Requerimiento 2:

O(n^2)

El requerimiento 2 tiene su carga independiente. En esta verifica si en la columna hay datos, después separa los jugadores que hay para que estén separados para las llaves. Después se busca en el árbol el nodo con la llave designada y si esta no existe se inserta y crea un nuevo valor como lista de tipo ARRAY.

En nuestro requerimiento extraemos los records del jugador gracias a om.get(), estos quedan almacenados en una lista. Después, consultamos el size de dicha lista para saber cuantos records tiene dicho jugador. Lo último que realizamos es organizar dicha lista con insertion sort y extraemos los 5 records del jugador. Claramente si tiene menos, la función solo retorna los que posea.

```
#=^..^=   [Requerimiento 2]  =^..^=    =^..^=    =^..^=    =^..^=

def R2_player_records(analyzer, player):
    existRecords = om.get(analyzer['Req2_RegistersByShorterTime'], player)   #O(logn)
    #contención de error
    if (existRecords is None):
        return None
    list_values = existRecords['value']                                      #O(1)
    Num_records = listSize(list_values)                                      #O(1)
    final_list = sortList(list_values, cmp_Req2)                             #O(n^2)
    if lt.size(final_list) >= 5:
        first_five_players = subList(final_list, 1, 5)                       #O(1)
        return first_five_players, Num_records, listSize(final_list)
    else:
        return final_list, Num_records, listSize(final_list)                 #La complejidad se resume en el ordenamiento con insertion
                                                                             #O(n^2)
```

Requerimiento 3:

O(n^2)

En el requerimiento 3 su carga independiente verifica si en la columna tiene datos, busca en el árbol el nodo con la llave designada y busca en el árbol el nodo con la llave designada. Si esta no existe se inserta y crea un nuevo valor como lista de tipo ARRAY. De lo contrario obtiene el valor y lo añade a su lista correspondiente. La complejidad de este se resume en su última instrucción (anteriormente mencionada) y es de 0(n).

Como tal en la función del requerimiento en la variable map_analyzer extraemos los valores del árbol, seguido a esto creamos una lista tipo ARRAY_LIST. Iteramos los valores del árbol para extraer los juegos y seguido a esto iteramos cada juego para obtener los datos de cada columna.
Si el juego cumple con la condición dada por el if después del for se añade este a la lista.
Lo último que realizamos es organizar dicha lista con insertion sort y extraemos los primeros y últimos.

```
#=^..^=   [Requerimiento 3]  =^..^=    =^..^=    =^..^=    =^..^=

def Req3_FastestRegistersByAttempts(analyzer, min_time, max_time):
    map_analyzer = analyzer["Req3_FastestRegistersByAttempts"]
    list_ = om.values(map_analyzer, min_time, max_time)                    #O(logn)
    fastest_records = lt.newList(datastructure = "ARRAY_LIST")             #O(1)

    for record in lt.iterator(list_):                                      #O(n^2)
        for finalRecord in lt.iterator(record):
            lt.addLast(fastest_records, finalRecord)                       #O(1)

    sorted_list = sortList(fastest_records, cmp_Req3and4)                  #O(n^2)
    final_list = FirstandLast(sorted_list)                                 #O(n^2)
    return final_list, listSize(sorted_list)                               #La complejidad se resume en el ordenamiento con insertion O(
```

Requerimiento 4:

Complejidad: O(n^2)

En el requerimiento 4 en la variable de map_analyzer hacemos alusión a el analyzer donde está el índice con la carga de datos para nuestro requerimiento 4. La complejidad de llamar esta carga podría resumirse en ella misma. La complejidad para la carga de datos es de 0(n) ya que la última instrucción es que, si la llave existe, obtenga el valor y lo añada a la lista.
En nuestro requerimiento extraemos los valores del árbol, seguido a esto creamos una lista tipo ARRAY_LIST. Iteramos los valores del árbol para extraer los record_date y seguido a esto iteramos cada final_record para obtener los datos de cada columna. Si el record está entonces se añade a la lista antes creada.
Lo último que realizamos es organizar dicha lista con insertion sort y extraemos los primeros y últimos.

```
#=^..^=   [Requerimiento 4]  =^..^=    =^..^=    =^..^=    =^..^=

def Req4_SlowRegistersbyDates(analyzer, min_date, max_date):
    map_analyzer = analyzer["Req4_SlowRegistersbyDates"]
    lst = om.values(map_analyzer, min_date, max_date)                    #O(logn)

    date_records = lt.newList(datastructure = "ARRAY_LIST")              #O(1)
    for record_date in lt.iterator(lst):                                 #O(n^2)
        for final_record in lt.iterator(record_date):
            lt.addLast(date_records, final_record)                       #O(1)

    sorted_list= sortList(date_records, cmp_Req3and4)                    #O(n^2)
    final_list = FirstandLast(sorted_list)                               #O(n^2)
    return final_list, listSize(sorted_list)                             #La complejidad se resume en el ordenamiento con insertion O(n
```

Requerimiento 5:

Complejidad : O(n^2)

En el requerimiento 5 en la variable de map_analyzer hacemos alusión a el analyzer donde está el índice con la carga de datos para nuestro requerimiento 5. La complejidad de llamar esta carga podría resumirse en ella misma. La complejidad para la carga de datos es de 0(n) ya que la última instrucción es que, si la llave existe, obtenga el valor y lo añada a la lista.

En nuestro requerimiento extraemos los valores del árbol, seguido a esto creamos una lista tipo ARRAY_LIST. Iteramos los valores del árbol para extraer los record_time y seguido a esto iteramos cada final_record para obtener los datos de cada columna.
Si el record está entonces se añade a la lista antes creada.
Lo último que realizamos es organizar dicha lista con insertion sort y extraemos los primeros y últimos.

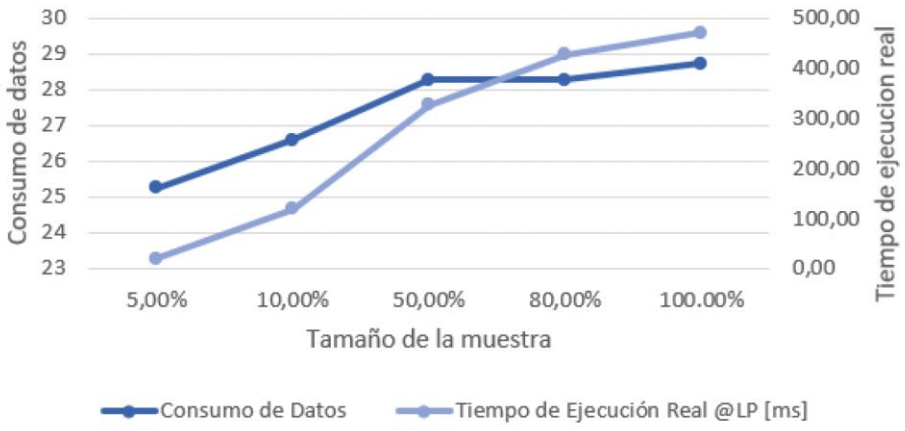
```
#=^..^=   [Requerimiento 5]  =^..^=    =^..^=    =^..^=    =^..^=

def Req5_RecentRegistersRecord(analyzer, min_time, max_time):
    map_analyzer = analyzer["Req5_RecentRegistersRecord"]
    lst = om.values(map_analyzer, min_time, max_time)                    #O(logn)

    time_records = lt.newList(datastructure = "ARRAY_LIST")              #O(1)
    for record_time in lt.iterator(lst):                                 #O(n^2)
        for final_record in lt.iterator(record_time):
            lt.addLast(time_records, final_record)                       #O(1)

    sorted_list = sortList(time_records, cmp_Req5)                       #O(n^2)
    final_list = FirstandLast(sorted_list)                               #O(n^2)
    return final_list, listSize(sorted_list)                             #La complejidad se resume en el ordenamiento con insertion O(n
```

Requerimiento 6



Requermiento 7